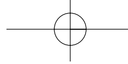
Johnson_02.qxd 4/1/05 4:56 PM Page 19



CHAPTER TWO

Kernel Overview

By Badari Pulavarti

INTRODUCTION

Now that you've made important decisions about how to install Linux on your system, you need to learn more about the Linux kernel to make important tuning decisions. We'll discuss how Linux evolved and then delve into its architecture. We'll include information on how the kernel is organized, what its responsibilities are, and how memory management is handled. We'll discuss process management and interprocess communication, followed by an overview of the Linux Symmetrical Multiprocessing Model. Finally, we'll examine the Linux file systems.

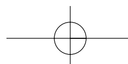
THE EVOLUTION OF LINUX

Linux is an operating system for personal computers developed by Linus Torvalds in 1991. Initially, Linux supported only the Intel 80x86 processor. Over the years, support has been added so that Linux can run on various other processors. Currently, Linux is one of very few operating systems that run on a wide range of processors, including Intel IA-32, Intel IA-64, AMD, DEC, PowerPC, Motorola, SPARC, and IBM S/390.

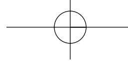
Linux is similar to UNIX in that it borrows many ideas from UNIX and implements the UNIX API. However, Linux is not a direct derivative of any particular UNIX distribution.

Linux is undoubtedly the fastest-growing operating system today. It is used in areas such as embedded devices all the way to mainframes. One of the interesting and most important facts about Linux is that it is open-sourced. The Linux kernel is licensed under the GNU General Public License (GPL); the kernel source code is freely available and can be modified to suit the needs of your machine.

As we move on to the next section, we'll take a more comprehensive look at the architecture of the Linux kernel.



Johnson_02.qxd 4/1/05 4:56 PM Page 20



LINUX KERNEL ARCHITECTURE

Let's begin this section by discussing the architecture of the Linux kernel, including responsibilities of the kernel, its organization and modules, services of the kernel, and process management.

Kernel Responsibilities

The kernel (also called the operating system) has two major responsibilities:

- To interact with and control the system's hardware components
- To provide an environment in which applications can run

Some operating systems allow applications to directly access hardware components, although this capability is very uncommon nowadays. UNIX-like operating systems hide all the low-level hardware details from an application. If an application wants to make use of a hardware resource, it must make a request to the operating system. The operating system then evaluates the request and interacts with the hardware component on behalf of the application, but only if it's valid. To enforce this kind of scheme, the operating system needs to depend on hardware capabilities that forbid applications to directly interact with them.

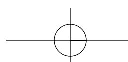
Organization and Modules

Like many other UNIX-like operating systems, the Linux kernel is *monolithic*. This means that even though Linux is divided into subsystems that control various components of the system (such as memory management and process management), all of these subsystems are tightly integrated to form the whole kernel. In contrast, *microkernel* operating systems provide bare, minimal functionality, and all other operating system layers are performed on top of microkernels as processes. Microkernel operating systems are generally slower due to message passing between the various layers. However, microkernel operating systems can be extended very easily.

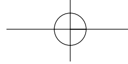
Linux kernels can be extended by *modules*. A module is a kernel feature that provides the benefits of a microkernel without a penalty. A module is an object that can be linked to the kernel at runtime.

Using Kernel Services

The kernel provides a set of interfaces for applications running in user mode to interact with the system. These interfaces, also known as system calls, give applications access to hardware and other kernel resources. System calls not only provide applications with abstracted hardware, but also ensure security and stability.



Johnson_02.qxd 4/1/05 4:56 PM Page 21



Most applications do not use system calls directly. Instead, they are programmed to an application programming interface (API). It is important to note that there is no relation between the API and system calls. APIs are provided as part of libraries for applications to make use of. These APIs are generally implemented through the use of one or more system calls.

/proc File System—External Performance View

The /proc file system provides the user with a view of internal kernel data structures. It also lets you look at and change some of the kernel internal data structures, thereby changing the kernal's behavior. The /proc file system provides an easy way to fine-tune system resources to improve the performance not only of applications but of the overall system.

/proc is a virtual file system that is created dynamically by the kernel to provide data. It is organized into various directories. Each of these directories corresponds to tunables for a given subsystem. Appendix A explains in detail how to use the /proc file system to fine-tune your system.

Another essential of the Linux system is memory management. In the next section, we'll cover five aspects of how Linux handles this management.

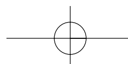
Memory Management

The various aspects of memory management in Linux include address space, physical memory, memory mapping, paging, and swapping.

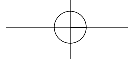
Address Space

One of the advantages of virtual memory is that each process thinks it has all the address space it needs. The virtual memory can be many times larger than the physical memory in the system. Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other. A process running one application cannot affect another, and the applications are protected from each other. The virtual address space is mapped to physical memory by the operating system. From an application point of view, this address space is a flat linear address space. The kernel, however, treats the user virtual address space very differently.

The linear address space is divided into two parts: user address space and kernel address space. The user address space cannot change every time a context switch occurs and the kernel address space remains constant. How much space is allocated for user space and kernel space depends mainly on whether the system is a 32-bit or 64-bit architecture. For example, x86 is a 32-bit architecture and supports only a 4GB address space. Out of this 4GB, 3GB is reserved for user space and 1GB is reserved for the kernel. The location of the split is determined by the PAGE_OFFSET kernel configuration variable.



Johnson_02.qxd 4/1/05 4:56 PM Page 22



Physical Memory

Linux uses an architecture-independent way of describing physical memory in order to support various architectures.

Physical memory can be arranged into banks, with each bank being a particular distance from the processor. This type of memory arrangement is becoming very common, with more machines employing NUMA (Nonuniform Memory Access) technology. Linux VM represents this arrangement as a *node*. Each node is divided into a number of blocks called *zones* that represent ranges within memory. There are three different zones: *ZONE_DMA*, *ZONE_NORMAL*, and *ZONE_HIGHMEM*. For example, x86 has the following zones:

ZONE_ DMA	First 16MB of memory
ZONE_ NORMAL	16MB – 896MB
ZONE_ HIGHMEM	896MB – end

Each zone has its own use. Some of the legacy ISA devices have restrictions on where they can perform I/O from and to. ZONE_DMA addresses those requirements.

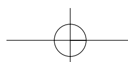
ZONE_NORMAL is used for all kernel operations and allocations. It is extremely crucial for system performance.

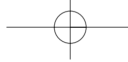
ZONE_ HIGHMEM is the rest of the memory in the system. It's important to note that ZONE_HIGHMEM cannot be used for kernel allocations and data structures—it can only be used for user data.

Memory Mapping

While looking at how kernel memory is mapped, we will use x86 as an example for better understanding. As mentioned earlier, the kernel has only 1GB of virtual address space for its use. The other 3GB is reserved for the kernel. The kernel maps the physical memory in ZONE_DMA and ZONE_NORMAL directly to its address space. This means that the first 896MB of physical memory in the system is mapped to the kernel's virtual address space, which leaves only 128MB of virtual address space. This 128MB of virtual space is used for operations such as vmalloc and kmap.

This mapping scheme works well as long as physical memory sizes are small (less than 1GB). However, these days, all servers support tens of gigabytes of memory. Intel has added PAE (Physical Address Extension) to its Pentium processors to support up to 64GB of physical memory. Because of the preceding memory mapping, handling physical memories in tens of gigabytes is a major source of problems for x86 Linux. The Linux kernel handles high memory (all memory about 896MB) as follows: When the Linux kernel needs to address a page in high memory, it maps that page into a small





virtual address space (kmap) window, operates on that page, and unmaps the page. The 64-bit architectures do not have this problem because their address space is huge.

Paging

Virtual memory is implemented in many ways, but the most effective way is hardware-based. Virtual address space is divided into fixed-size chunks called *pages*. Virtual memory references are translated into addresses in physical memory using page tables. To support various architectures and page sizes, Linux uses a three-level paging mechanism. The three types of page tables are as follows:

- Page Global Directory (PGD)
- Page Middle Directory (PMD)
- Page Table (PTE)

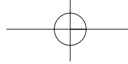
Address translation provides a way to separate the virtual address space of a process from the physical address space. Each page of virtual memory can be marked "present" or "not present" in the main memory. If a process references an address in virtual memory that is not present, hardware generates a page fault, which is handled by the kernel. The kernel handles the fault and brings the page into main memory. In this process, the system might have to replace an existing page to make room for the new one.

The replacement policy is one of the most critical aspects of the paging system. Linux 2.6 fixed various problems surrounding the page selection and replacement that were present in previous versions of Linux.

Swapping

Swapping is the moving of an entire process to and from secondary storage when the main memory is low. Many modern operating systems, including Linux, do not use this approach, mainly because context switches are very expensive. Instead, they use paging. In Linux, swapping is performed at the page level rather than at the process level. The main advantage of swapping is that it expands the process address space that is usable by a process. As the kernel needs to free up memory to make room for new pages, it may need to discard some of the less frequently used or unused pages. Some of the pages cannot be freed up easily because they are not backed by disks. Instead, they have to be copied to a backing store (swap area) and need to be read back from the backing store when needed. One major disadvantage of swapping is speed. Generally, disks are very slow, so swapping should be eliminated whenever possible.

Johnson_02.qxd 4/1/05 4:56 PM Page 24



PROCESS MANAGEMENT

This section discusses process management in Linux, including processes, tasks, kernel threads, scheduling, and context switching.

Processes, Tasks, and Kernel Threads

A *task* is simply a generic "description of work that needs to be done," whether it is a lightweight thread or a full process.

A *thread* is the most lightweight instance of a task. Creating a thread in the kernel can be expensive or relatively cheap, depending on the characteristics the thread needs to possess. In the simplest case, a thread shares everything with its parent including text, data, and many internal data structures, possessing only the minimum differences necessary to distinguish the thread from another thread.

A *process* is a "heavier" data structure in Linux. Several threads can operate within (and share some resources of) a single process, if desired. In Linux, a process is simply a thread with all of its heavyweight characteristics. Threads and processes are scheduled identically by the scheduler.

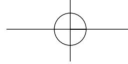
A *kernel thread* is a thread that always operates in kernel mode and has no user context. Kernel threads are usually present for a specific function and are most easily handled from within the kernel. They often have the desirable side effect of being schedulable like any other process and of giving other processes a target (by sending a signal) when they need that function to take effect.

Scheduling and Context Switching

Process scheduling is the science (some would say art) of making sure that each process gets a fair share of the CPU. There is always an element of disagreement over the definition of "fair" because the choices the scheduler must make often depend on information that is not apparent.

Process scheduling is covered more thoroughly in later chapters in this book, but it is important to note that it is deemed by many Linux users to be more important to have a scheduler that gets it mostly right all of the time than completely right most of the time—that is, slow-running processes are better than processes that stop dead in their tracks either due to deliberate choices in scheduling policies or outright bugs. The current Linux scheduler code adheres to this principle.

When one process stops running and another replaces it, this is known as a *context switch*. Generally, the overhead for this is high, and kernel programmers and application programmers try to minimize the number of context switches performed by the system. Processes can stop running voluntarily because they are waiting for some event or



resource, or involuntarily if the system decides it is time to give the CPU to another process. In the first case, the CPU may actually become idle if no other process is waiting to run. In the second case, either the process is replaced with another that has been waiting, or the process is given a new *timeslice*, or period of time in which to run, and is allowed to continue.

Even when processes are being scheduled and run in an orderly fashion, they can be interrupted for other, higher-priority tasks. If a disk has data ready from a disk read, it signals the CPU and expects to have the information taken from it. The kernel must handle this situation in a timely fashion, or it will slow down the disk's transfer rates. Signals, interrupts, and exceptions are asynchronous events that are distinct but similar in many ways, and all must be dealt with quickly, even when the CPU is already busy.

For instance, a disk with data ready causes an *interrupt*. The kernel calls the interrupt handler for that particular device, interrupting the process that is currently running, and utilizing many of its resources. When the interrupt handler is done, the currently running process resumes. This in effect *steals* time from the currently running process, because current versions of the kernel measure only the time that has passed since the process was placed on the CPU, ignoring the fact that interrupts can use up precious milliseconds for that process.

Interrupt handlers are usually very fast and compact and thereby handle and clear interrupts quickly so that the next bit of data can come in. At times, however, an interrupt can require more work than is prudent in the short time desired in an interrupt handler. An interrupt can also require a well-defined environment to complete its work (remember, an interrupt utilizes a random process's resources). In this case, enough information is collected to defer the work to what is called a *bottom half handler*. The bottom half handler is scheduled to run every so often. Although the use of bottom halves was common in earlier versions of Linux, their use is discouraged in current versions of Linux.

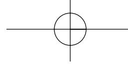
Interprocess Communications

Linux supports a number of Interprocess Communication (IPC) mechanisms to allow processes to communicate with each other. Signals and pipes are the basic mechanisms, but Linux also supports System V IPC mechanisms.

Signals

Signals notify events to one or more processes and can be used as a primitive way of communication and synchronization between user processes. Signals can also be used for job control.

Johnson_02.qxd 4/1/05 4:56 PM Page 26



The kernel can generate a set of defined signals, or they can be generated by other processes in the system, provided that they have the correct privileges.

Processes can choose to ignore most of the signals that are generated, with two exceptions: SIGSTOP and SIGKILL. The SIGSTOP signal causes a process to halt its execution. The SIGKILL signal causes a process to exit and be ignored. With the exception of the SIGSTOP and SIGKILL signals, a process can choose how it wants to handle the various signals. Processes can block the signals, or they can either choose to handle the signals themselves or allow the kernel to handle the signals. If the kernel handles the signals, it performs the default actions required for the signal. Linux also holds information about how each process handles every possible signal.

Signals are not delivered to the process as soon as they are generated, but instead are delivered when the process resumes running. Every time a process exits a system call, if there are any unblocked signals, the signals are then delivered.

Linux is POSIX-compatible so the process can specify which signals are blocked when a particular signal-handling routine is called.

Pipes

A *pipe* is a unidirectional, first-in first-out (FIFO), unstructured stream of data. Writers add data to one end of the pipe, and readers get the data from other end of the pipe. After the data is read, it is removed from the pipe. Pipes provide simple flow control.

For example, the following command pipes output from the `ls` command, which lists the directory's files, into the standard input of the `less` command, which paginates the files:

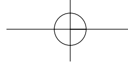
```
$ ls | less
```

Linux also supports *named pipes*. Unlike pipes, named pipes are not temporary objects; they are entities in the file system and can be created using the `mkfifo` command.

System V IPC Mechanisms

Linux supports three types of interprocess communication mechanisms that first appeared in UNIX System V (1983). These mechanisms are message queues, semaphores, and shared memory. The mechanisms all share common authentication methods. Processes can access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions much like accesses to files are checked. The access rights to a System V IPC object are set by the creator of the object via system calls.

Johnson_02.qxd 4/1/05 4:56 PM Page 27



Message Queues

Message queues allow one or more processes to write messages, which will be read by one or more reading processes. In terms of functionality, message queues are equivalent to pipes, but message queues are more versatile than pipes and have several advantages over pipes. Message queues pass data in messages rather than as an unformatted stream of bytes, allowing data to be processed easily. The messages can be associated with a type, so the receiver can check for urgent messages before processing non-urgent messages. The type field can also be used to designate a recipient in case multiple processes share the same message queue.

Semaphores

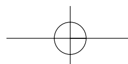
Semaphores are objects that support two atomic operations: set and test. Semaphores can be used to implement various synchronization protocols. Semaphores can best be described as counters that control access to shared resources by multiple processes. Semaphores are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on the resource.

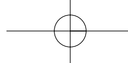
A problem with semaphores, called *deadlocking*, occurs when one process has altered a semaphore's value as it enters a critical region but then fails to leave the critical region because it crashed or was killed. Linux protects against this by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphore will be returned to the state it was in before the process's set of semaphore operations were applied.

Shared Memory

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. Access to shared memory areas is controlled through keys and access rights checking. When the memory is being shared, there are no checks on how the processes are using the memory. Each process that wishes to share the memory must attach to that virtual memory via a system call. The process can choose where in its virtual address space the shared memory goes, or it can let Linux choose a free area large enough. The first time that a process accesses one of the pages of the shared virtual memory, a page fault occurs. When Linux fixes that page fault, it allocates a physical page and creates a page table entry for it. Thereafter, access by the other processes causes that page to be added to their virtual address spaces.

When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory, the detach operation affects only the current process. When the last process sharing the memory detaches from it, the pages of the shared memory currently in physical memory are freed.





Further complications arise when shared virtual memory is not locked into physical memory. In this case, the pages of the shared memory may be swapped out to the system's swap disk during periods of high memory usage.

The Linux Symmetrical Multiprocessing (SMP) Model

Types of Multiprocessing

A multiprocessing system consists of a number of processors communicating via a bus or a network. There are two types of multiprocessing systems: loosely coupled and tightly coupled.

Loosely coupled systems consist of processors that operate stand-alone. Each processor has its own bus, memory, and I/O subsystem, and communicates with other processors through the network medium. Loosely coupled systems can be either homogeneous or heterogeneous.

Tightly coupled systems consist of processors that share the memory, bus, devices, and sometimes cache. Tightly coupled systems run a single instance of the operating system. Tightly coupled systems can be classified into symmetric and asymmetric systems. Asymmetric systems are configured so that each processor is assigned a specific task. Asymmetric systems have a single "master" processor that controls all others. Symmetric systems treat all processors the same way—processes have equal access to all system resources. In the symmetric model, all tasks are spread equally across all processors.

Symmetric systems are subdivided into further classes consisting of dedicated and shared cache systems. Symmetrical Multiprocessing (SMP) systems have become very popular and have become the default choice for many large servers.

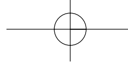
Concurrency and Data Serialization

In an ideal world, an SMP system with *n* processors would perform *n* times better than a uniprocessor system. In reality, this is not the case. The main reason that no SMP system is 100% scalable is because of the overhead involved in maintaining additional processors.

Locks, Lock Granularity, and Locking Overhead

Locks basically protect multiple threads from accessing or modifying a piece of critical information at the same time. Locks are especially used on SMP systems where multiple processors execute multiple threads at the same time. The problem with locking is that if two or more processors are competing for the same lock at the same time, only one is granted the lock, and the other waits, spinning, for the lock to be released. In other

Johnson_02.qxd 4/1/05 4:56 PM Page 29



words, the other processors are not really doing any useful work. Locking, therefore, must be limited to the smallest amount of time possible.

Another common technique used to address this problem is to employ finer-grain locking. With finer-grain locking, instead of using a single lock to protect 100 things, 100 locks are used instead. Although the concept seems very simple, most of the time, it is hard to implement because of various interactions, dependencies, and deadlock. You need to program methodically to prevent deadlock situations, compared to having a single lock.

Another important area to consider is locking overhead. All locking techniques come with a price. Operating system designers need to choose the right kind of locking primitive to address a rights issue. In Linux 2.6, most global locks are removed and most of the locking primitives are optimized for extremely low overhead.

Cache Coherency

Cache coherency is a problem that occurs in multiprocessors, because each processor has an individual cache, and multiple copies of certain data exist in the system. When the data is changed, only one processor's cache has the new value. All other processors' cache has old values.

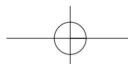
Processor Affinity

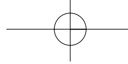
Processor affinity is one of the most important things that can improve system performance. As processes access various resources in the system, lots of information about the resources will be in processor caches, so it's better for a processor to run on the same processor due to the cache warmth. In some architectures, especially with NUMA, some resources are closer to the processor compared to others in the same system. In these systems, processor affinity is extremely important for system performance.

The file system is one of the most important parts of an operating system. In the following sections, the Linux alternative in file systems explains how well Linux has you covered.

File Systems

This section provides an overview of file systems on Linux and discusses the virtual file system, the ext2 file system, LVM and RAID, volume groups, device special files, and devfs.





Virtual File System (VFS)

One of the most important features of Linux is its support for many different file systems. This makes it very flexible and well able to coexist with many other operating systems. Virtual file system (VFS) allows Linux to support many, often very different, file systems, each presenting a common software interface to the VFS. All of the details of the Linux file systems are translated by software so that all file systems appear identical to the rest of the Linux kernel and to programs running in the system. The Linux Virtual File System layer allows you to transparently mount many different file systems at the same time.

The Linux virtual file system is implemented so that access to its files is as fast and efficient as possible. It must also make sure that the files and their data are maintained correctly.

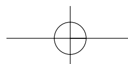
ext2fs

The first file system that was implemented on Linux was ext2fs. This file system is the most widely used and the most popular. It is highly robust compared to other file systems and supports all the normal features a typical file system supports, such as the capability to create, modify, and delete file system objects such as files, directories, hard links, soft links, device special files, sockets, and pipes. However, a system crash can leave an ext2 file system in an inconsistent state. The entire file system has to be validated and corrected for inconsistencies before it is remounted. This long delay is sometimes unacceptable in production environments and can be irritating to the impatient user. This problem is solved with the support of journaling. A newer variant of ext2, called the ext3 file system, supports journaling. The basic idea behind journaling is that every file system operation is logged before the operation is executed. Therefore, if the machine dies between operations, only the log needs to be replayed to bring the file system back to consistency.

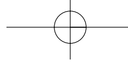
LVM and RAID

Volume managers provide a logical abstraction of a computer's physical storage devices and can be implemented for several reasons. On systems with a large number of disks, volume managers can combine several disks into a single logical unit to provide increased total storage space as well as data redundancy. On systems with a single disk, volume managers can divide that space into multiple logical units, each for a different purpose. In general, a volume manager is used to hide the physical storage characteristics from the file systems and higher-level applications.

Redundant Array of Inexpensive Disks (RAID) is a type of volume management that is used to combine multiple physical disks for the purpose of providing increased I/O throughput or improved data redundancy. There are several RAID levels, each



Johnson_02.qxd 4/1/05 4:56 PM Page 31



providing a different combination of the physical disks and a different set of performance and redundancy characteristics. Linux provides four different RAID levels:

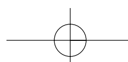
- *RAID-Linear* is a simple concatenation of the disks that comprise the volume. The size of this type of volume is the sum of the sizes of all the underlying disks. This RAID level provides no data redundancy. If one disk in the volume fails, the data stored on that disk is lost.
- *RAID-0* is simple striping. Striping means that as data is written to the volume, it is interleaved in equal-sized "chunks" across all disks in the volume. In other words, the first chunk of the volume is written to the first disk, the second chunk of the volume is written to the second disk, and so on. After the last disk in the volume is written to, it cycles back to the first disk and continues the pattern. This RAID level provides improved I/O throughput.
- *RAID-1* is mirroring. In a mirrored volume, all data is replicated on all disks in the volume. This means that a RAID-1 volume created from *n* disks can survive the failure of n–1 of those disks. In addition, because all disks in the volume contain the same data, reads to the volume can be distributed among the disks, increasing read throughput. On the other hand, a single write to the volume generates a write to each of the disks, causing a decrease in write throughput. Another downside to RAID-1 is the cost. A RAID-1 volume with *n* disks costs *n* times as much as a single disk but only provides the storage space of a single disk.
- *RAID-5* is striping with parity. This is similar to RAID-0, but one chunk in each stripe contains parity information instead of data. Using this parity information, a RAID-5 volume can survive the failure of any single disk in the volume. Like RAID-0, RAID-5 can provide increased read throughput by splitting large I/O requests across multiple disks. However, write throughput can be degraded, because each write request also needs to update the parity information for that stripe.

Volume Groups

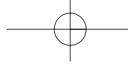
The concept of volume-groups (VGs) is used in many different volume managers.

A volume-group is a collection of disks, also called physical-volumes (PVs). The storage space provided by these disks is then used to create logical-volumes (LVs).

The main benefit of volume-groups is the abstraction between the logical- and physical-volumes. The VG takes the storage space from the PVs and divides it into fixed-size chunks called physical-extents (PEs). An LV is then created by assigning one or more PEs to the LV. This assignment can be done in any arbitrary order—there is no dependency on the underlying order of the PVs, or on the order of the PEs on a particular PV.



Johnson_02.qxd 4/1/05 4:56 PM Page 32



This allows LVs to be easily resized. If an LV needs to be expanded, any unused PE in the group can be assigned to the end of that LV. If an LV needs to be shrunk, the PEs assigned to the end of that LV are simply freed.

The volume-group itself is also easily resizeable. A new physical-volume can be added to the VG, and the storage space on that PV becomes new, unassigned physical-extents. These new PEs can then be used to expand existing LVs or to create new LVs. Also, a PV can be removed from the VG if none of its PEs are assigned to any LVs.

In addition to expanding and shrinking the LVs, data on the LVs can be "moved" around within the volume-group. This is done by reassigning an extent in the LV to a different, unused PE somewhere else in the VG. When this reassignment takes place, the data from the old PE is copied to the new PE, and the old PE is freed.

The PVs in a volume-group do not need to be individual disks. They can also be RAID volumes. This allows a user to get the benefit of both types of volume management. For instance, a user might create multiple RAID-5 volumes to provide data redundancy, and then use each of these RAID-5 volumes as a PV for a volume-group. Logical-volumes can then be created that span multiple RAID-5 volumes.

Device Special Files

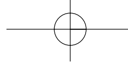
A typical Linux system has at least one hard disk, a keyboard, and a console. These devices are handled by their corresponding device drivers. However, how would a user-level application access the hardware device? Device special files are an interface provided by the operating system to applications to access the devices. These files are also called device nodes that reside in the /dev directory. The files contain a major and minor number pair that identifies the device they support. Device special files are like normal files with a name, ownership, and access permissions.

There are two kinds of device special files: block devices and character devices. Block devices allow block-level access to the data residing on the device, and character devices allow character-level access to the device. When you issue the `ls -l` command on a device, if the returned permission string starts with a b, it is a block device; if it starts with a c, it is a character device.

devfs

The virtual file system, devfs, manages the names of all the devices. devfs is an alternative to the special block and character device node that resides on the root file system. devfs reduces the system administrative task of creating device nodes for each device in the system. This job is automatically handled by devfs. Device drivers can register devices to devfs through device names instead of through the traditional major-minor number scheme. As a result, the device namespace is not limited by the number of major and minor numbers.

Johnson_02.qxd 4/1/05 4:56 PM Page 33



A system administrator can mount the devfs file system many times at different mount points, but changes to a device node are reflected on all the device nodes on all the mount points. Also, the devfs namespace exists in the kernel even before it is mounted. Essentially, this makes the availability of device nodes independent of the availability of the root file system.

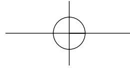
With the traditional solution, a device node is created in the /dev directory for each and every conceivable device in the system, irrespective of the existence of the device. However, in devfs, only the necessary and sufficient device entries are maintained.

NEW FEATURES IN LINUX 2.6

Linux is getting better! New features are constantly being added to ensure its integrity and increase its functionality. The following is a list of key new features in Linux 2.6. Note that some of these features have been backported to various 2.4-based distribution releases:

- New architecture support (x86-64)
- Hyperthreading/SMT support
- (O)1 scheduler
- Preemption support
- NUMA
- NPTL
- I/O scheduler support
- Block layer rewrite
- AIO
- New file systems
- ACL
- Sysfs
- Udev
- Networking improvements
- Linux security module
- Processor affinity
- Module rewrite
- Unified driver model
- LVM

Johnson_02.qxd 4/1/05 4:56 PM Page 34



SUMMARY

In this chapter, we've explained the key components of the Linux kernel to help you understand how the kernel is architected and organized. In the next chapter, we'll go a step further and discuss some of the servers that Linux runs on.

REFERENCES

[1] Gorman, Mel. *Understanding the Linux Virtual Memory Manager*, Prentice Hall, 2004.

[2] Love, Robert. "Linux Kernel Development," *Developer's Library*, 2004.

[3] Vahalia, Uresh. *UNIX Internals: The New Frontiers*, Prentice Hall, 1996.

[4] Linux kernel mailing lists.

[5] Linux kernel sources and documentation.